PRÁCTICA 1:

SISTEMA DE AVIACIÓN

- Juan Manuel Valenzuela González – amcgil@uma.es
- Artur Vargas Carrión – arturvargas797@uma.es
- Jorge Repullo Serrano – jorgers4@uma.es
- Eduardo González Bautista – edugb@uma.es
- Rubén Oliva Zamora – rubenoliva@uma.es
- David Muñoz del Valle – davidmunvalle@uma.es
- Alejandro Jiménez González – alejg411@uma.es

Modelado y Diseño del Software

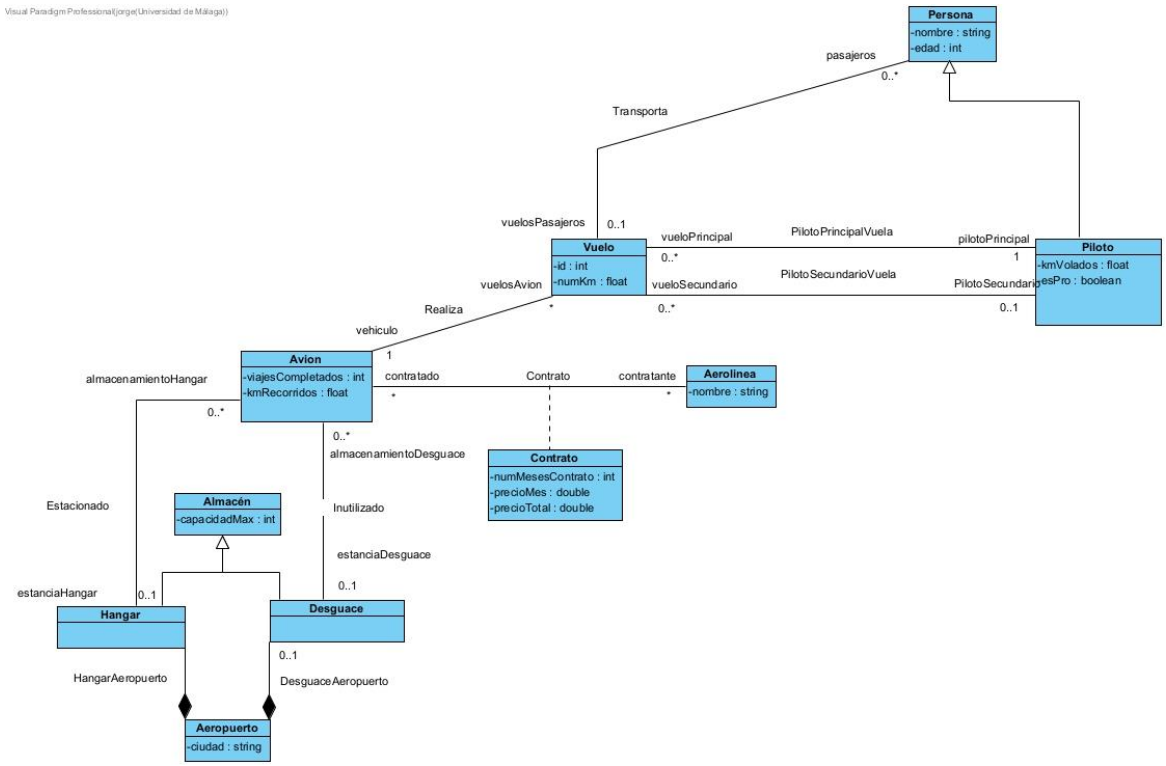
GR1-3 | Ingeniería del Software A | Universidad de Málaga

ÍNDICE DE CONTENIDO

1. Visual Paradigm

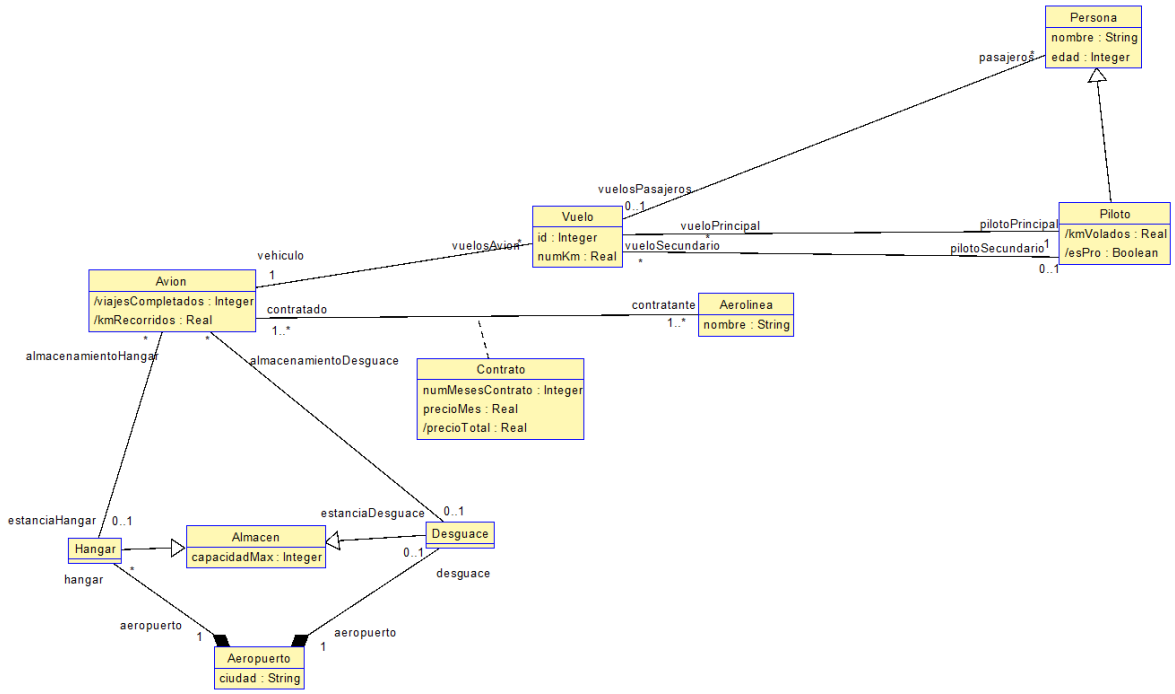
Se ha utilizado Visual Paradigm (VP) para realizar el **modelado conceptual base** (primera aproximación) del sistema de aviación. Este entorno nos ha permitido crear un primer diseño del sistema, identificando las entidades principales y las relaciones entre ellas, lo que nos ha permitido iterar y refinar el modelo hasta llegar a la versión final. VP nos ha sido de gran utilidad para visualizar y entender de forma clara la estructura del sistema antes de pasar a las fases de validación y verificación.

El sistema que hemos modelado recopila información sobre aviones, aerolíneas, vuelos, pilotos y pasajeros, gestionando aspectos clave como los vuelos completados por cada avión, los contratos entre aerolíneas y aviones, el estado de los aviones (volando, en hangar o desguace), y las estadísticas de los pilotos. Este modelo tiene como objetivo garantizar una representación fiel de las interacciones y restricciones que rigen el sistema de aviación.



1 Diagrama de clases del sistema de aviación en Visual Paradigm

2. USE



2 Diagrama de clases del sistema de aviación en USE

2.1. Aclaraciones sobre clases

Las decisiones sobre qué clases usar han sido bastante directas, ya que no han requerido de mucho más que una lectura y análisis exhaustivo del enunciado del problema. Tras desglosar el enunciado, **obtuvimos todas las clases salvo dos**, que se añadieron con el fin de **no repetir atributos comunes a otras clases**. Estas dos clases añadidas son **`Almacen`** y **`Persona`** (clases de generalización), ya que los pares de clases **`Hangar`** / **`Desguace`** / **`Piloto`** comparten atributos y así se evita la redundancia de atributos idénticos en clases distintas.

En el diseño original del sistema de aviación, teníamos una clase **`Persona`**, de la cual heredaban las clases **`Piloto`** y **`Pasajero`**. Esta estructura generaba un problema cuando un piloto deseaba viajar como pasajero, ya que el sistema requería crear una nueva instancia de la clase **`Pasajero`** para representar esa acción.

El inconveniente principal radicaba en la falta de un identificador único compartido entre las clases **`Piloto`** y **`Pasajero`**, lo que provocaba ambigüedad al intentar distinguir si se trataba de la misma persona o simplemente de otra persona con el mismo nombre y edad. Como resultado, existía el riesgo de duplicar instancias de la misma persona bajo diferentes roles, sin una forma clara de relacionarlas.

La solución implementada consistió en eliminar la clase **`Pasajero`** y redirigir todas sus relaciones directamente a la clase **`Persona`**. De este modo, los pilotos pasan a ser la única herencia de personas que pueden desempeñar diferentes roles, incluyendo el de pasajeros. Esto unifica la representación de los individuos bajo una única clase, **`Persona`**, independientemente del rol que desempeñen en un vuelo (piloto o pasajero).

2.2. Aclaraciones sobre relaciones y atributos

2.2.1. Relaciones

En primera instancia, en el uso de los **roles de las relaciones** se han usado estrictamente **sustantivos**, y en el **nombre de dichas relaciones** se han usado estrictamente **verbos**. Esto se debe a que a la hora de hacer llamadas a los roles de estas relaciones en USE, este patrón de uso facilita enormemente la comprensión al evitar tener que pensar en quien realiza la acción (roles) o qué significa exactamente la relación (nombres).

Durante el diseño de este sistema, nos surgieron varias formas de afrontar el problema. En primera instancia, nos planteamos crear una relación entre piloto y aerolínea después de leer esta parte del enunciado: "No se permite a un piloto trabajar o haber trabajado para más de dos aerolíneas diferentes". Finalmente, decidimos eliminar esta relación ya que podía dar lugar a **problemas de consistencia**, ya que ya se podía acceder a una relación entre piloto y aerolínea por este camino: **`Piloto` → `Vuelo` → `Avion` → `Aerolínea`**.
En cuanto a las clases **`Piloto`** y **`Vuelo`**, decidimos crear dos relaciones. En el enunciado se especifica que siempre debe haber un piloto principal, y que puede haber un piloto secundario. Dado que estas dos relaciones tendrían multiplicidad distinta, y que para que un piloto sea "pro" es condición necesaria que haya participado en 2000 vuelos como piloto secundario y en 1000 vuelos como principal, decidimos que lo mejor serían **dos relaciones distintas**.

Sobre la clase **`Avion`**, en primera instancia **planteamos el uso de un enumerado** para guardar el estado del avión (volando, en desguace o en hangar). Finalmente, **decidimos que lo mejor sería una relación con `Desguace` y `Hangar`**, ya que esto nos permitiría **saber en qué desguace o hangar se ubica**, o en el caso de que no esté en ninguno, nos permitiría saber que se encuentra volando en ese instante.

Las clases **`Hangar`** y **`Desguace`** tienen una relación de composición con la clase **`Aeropuerto`**, ya que en el enunciado de la práctica se especifica que **tanto los hangares como los desguaces están dentro de un aeropuerto**, por lo que no podrían existir sin la previa existencia de una instancia de esta clase que los contenga.

La multiplicidad de las relaciones **`Avion-Hangar`** y **`Avion-Desguace`** son 0..1, ya que de esa forma cumplimos la restricción de que **un avión solo puede estar en un hangar o en un desguace**, y no en varios a la vez. Lo mismo pasa con la relación **`Desguace-Aeropuerto`**.

Como se ha comentado anteriormente, la relación **`Piloto-Aerolinea`** no es directa, sino que se llega por el camino: **`Piloto` → `Vuelo` → `Avion` → `Aerolinea`**. En el enunciado se especifica que **un piloto no puede volar para más de dos aerolíneas** (invariante) pero **un avión sí que puede pertenecer a varias aerolíneas**. Al tener la relación así modelada, si un avión pertenece a varias aerolíneas y el piloto ha hecho un vuelo en ese avión, **el piloto trabaja para todas las aerolíneas a las que pertenece el avión**. El enunciado no deja claro que esto sea así, pero debido a que **todavía no usamos cuestiones temporales en el modelado** (guardar para que aerolínea volaba el avión cuando el piloto realizó el vuelo), pensamos que es la mejor forma de modelarlo. Esto nos lleva a que el invariante antes mencionado no se cumplirá si un avión pertenece a más de dos aerolíneas.

2.2.2. Atributos derivados

En cuanto a los **atributos derivados**, son aquellos cuyo valor no se almacena explícitamente en la base de datos, sino que se calcula a partir de otros atributos o relaciones en el modelo.
En nuestro código lo podemos ver en los siguientes atributos:

`Avion → viajesCompletados:` cuenta la cantidad de vuelos que ha realizado un avión. Se deriva contando el tamaño de la colección de vuelos asociados al avión a través de la relación **`Realiza`**.
`Avion → kmRecorridos:` suma los kilómetros recorridos por un avión. Se deriva sumando el atributo **`numKm`** de cada vuelo que ha realizado el avión, también a través de la relación **`Realiza.`**

```
class Avion
    attributes
        viajesCompletados : Integer derive : self.vuelosAvion->size()
        kmRecorridos : Real derive : self.vuelosAvion->collect(t|t.numKm)->sum()
end
```

`Contrato → precioTotal:` representa el costo total de un contrato entre una aerolínea y un avión, y se deriva multiplicando el precio mensual por el número de meses del contrato.

```
associationclass Contrato between
        Avion [1..*] role contratado
        Aerolinea [1..*] role contratante
    attributes
        numMesesContrato : Integer
        precioMes : Real
        precioTotal : Real derive : self.precioMes * self.numMesesContrato
end
```

`Piloto → kmVolados:` suma todos los kilómetros que ha volado un piloto, tanto como piloto principal como secundario. Se deriva sumando los kilómetros de los vuelos donde el piloto fue principal y los vuelos donde fue secundario, a través de las relaciones **`PilotoPrincipalVuela y PilotoSecundarioVuela.`**
`Piloto → esPro:` determina si un piloto es profesional. Se deriva evaluando si el piloto ha sido principal en al menos 1000 vuelos y secundario en al menos 2000 vuelos, también a través de las relaciones **`PilotoPrincipalVuela y PilotoSecundarioVuela.`**

```
class Piloto < Persona
    attributes
        kmVolados : Real derive : -- Los kmVolados de un piloto son la suma de los km volados como principal y como secundario
            self.vueloPrincipal->collect(v | v.numKm)->sum() +
            self.vueloSecundario->collect(v | v.numKm)->sum()
        esPro : Boolean derive : -- Cada PilotoPro debe tener al menos 1000 vuelos como Principal y 2000 como Secundario
            self.vueloPrincipal->size() >= 1000 and self.vueloSecundario->size() >= 2000
end
```

2.3. Invariantes

Para comprobar la eficacia de las invariantes hemos realizado dos archivos .soil, en uno de ellos nos hemos asegurado que **todos las invariantes funcionen a la perfección (P1_SistemaAviacionTodoBien.soil)** comprobando esto en el CMD mediante el comando **`Check`** que nos proporciona una lista indicando si estos funcionan correctamente **`OK`** o si por el contrario no se cumple la invariante **`FAILED.`**. Por otro lado, tenemos el otro archivo .soil que comprobaría lo contrario, **que todas las invariantes no se cumplan (P1_SistemaAviacionTodoMal.soil)**.
Para ver a detalle qué datos rompen las invariantes, por favor consúltese el archivo **P1_SistemaAviacionTodoMal.soil**, ya que en ese archivo aparecen, junto a comentarios explicativos, los datos que rompen cada uno de los invariantes

2.3.1. Explicación de invariantes

1. **`Aerolinea`** → **`NombreAerolineaDistinto:`** asegura que cada aerolínea tiene un nombre único. Se obtiene verificando que no hay dos instancias de **`Aerolínea`** que compartan el mismo valor de nombre.

```
inv NombreAerolineaDistinto :
    Aerolinea.allInstances()->isUnique(nombre)
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (1) `Aerolinea::NombreAerolineaDistinto': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (1) `Aerolinea::NombreAerolineaDistinto': FAILED.
```

2. **`Aeropuerto`** → **`CiudadAeropuertoDistinta`**: asegura que no haya más de un aeropuerto en la misma ciudad. Se constata verificando que cada ciudad en la que se ubican los aeropuertos tenga un nombre único entre todas las instancias de **`Aeropuerto`**.

```
inv CiudadAeropuertoDistinta :
    Aeropuerto.allInstances()->isUnique(ciudad)
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (2) `Aeropuerto::CiudadAeropuertoDistinta': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (2) `Aeropuerto::CiudadAeropuertoDistinta': FAILED.
```

3. **`Avion`** → **`AvionCorrectamenteDesguace_1000:`** establece que un avión solo puede ser destinado a un desguace si ha completado 1000 viajes. Se calculan evaluando si el número de **`viajesCompletados`** es menor que 1000 y asegurándose de que no está en un desguace, o si ha completado 1000 o más viajes, confirmando que sí está en un desguace.

```
inv AvionCorrectamenteDesguace_1000 :
    (self.viajesCompletados < 1000 and self.estanciaDesguace.oclIsUndefined()) or
    (self.viajesCompletados >= 1000 and (not self.estanciaDesguace.oclIsUndefined()))
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (3) `Avion::AvionCorrectamenteDesguace_1000': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (3) `Avion::AvionCorrectamenteDesguace_1000': FAILED.
```

4. **`Avion` → `AvionCorrectamenteDesguace_2`:** es una **prueba de la lógica de la invariante anterior**, permitiendo comprobar que un avión con menos de 2 viajes no esté en un desguace. Se verifica revisando si los **`viajesCompletados`** son menores que 2 y asegurando que el avión no esté en un desguace, o si tiene 2 o más viajes, debe estar en un desguace. Realmente **no se pide en el enunciado**, solo existe para evitarnos tener que meter 1000 aviones para poder comprobar esta invariante (**de ahí que esté comentado** y que no haya fotos con la salida del check abajo, pero funciona como es debido).

```
--inv AvionCorrectamenteDesguace_2 :
--    self.viajesCompletados < 2 and self.estanciaDesguace.oclIsUndefined()
--    or (self.viajesCompletados >= 2 and (not self.estanciaDesguace.oclIsUndefined()))
```

5. **`Avion` → `AvionUnicoEstado`:** garantiza que un avión se encuentre en un solo estado a la vez. Se determina asegurando que un avión no puede estar volando y también en un hangar o en un desguace simultáneamente; debe estar en uno de estos tres estados. Si no está ni en hangar ni en desguace, es que está volando (no hay otro posible estado).

```
inv AvionUnicoEstado :
    (self.estanciaHangar.oclIsUndefined() or self.estanciaDesguace.oclIsUndefined())
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (4) `Avion::AvionUnicoEstado': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (4) `Avion::AvionUnicoEstado': FAILED.
```

6. **`Desguace` → `CapacidadMaxDesguace`**: asegura que en un desguace no haya más aviones de los que puede albergar. Se revisa asegurando que el tamaño de la colección de aviones almacenados en el desguace no exceda el **`capacidadMax`** del mismo.

```
inv CapacidadMaxDesguace :
    self.almacenamientoDesguace -> size() <= self.capacidadMax
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (5) `Desguace::CapacidadMaxDesguace': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (5) `Desguace::CapacidadMaxDesguace': FAILED.
```

7. **`Hangar` → `CapacidadMaxHangar`**: similar a la invariante anterior, garantiza que en un hangar no haya más aviones de los que puede albergar. Se determina comprobando que el tamaño de la colección de aviones almacenados en el hangar no exceda el **`capacidadMax`** del mismo.

```
inv CapacidadMaxHangar :
    self.almacenamientoHangar -> size() <= self.capacidadMax
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (6) `Hangar::CapacidadMaxHangar': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (6) `Hangar::CapacidadMaxHangar': FAILED.
```

8. **`Piloto` → `maxDosAerolineasPorPiloto`**: limita la cantidad de aerolíneas para las que puede haber trabajado un piloto a un máximo de dos. Se extrae utilizando un conjunto que recopila las aerolíneas para las que ha volado como principal o secundario y verificando que su tamaño no exceda dos.

```
inv maxDosAerolineasPorPiloto:       -- las aerolineas para las que ha trabajado como principal o como secundario
    let aerolineasPilotadas: Set(Aerolinea) =   self.vueloPrincipal->collect(v | v.vehiculo.contratante)->asSet()->
                                                union(self.vueloSecundario->collect(v | v.vehiculo.contratante)->asSet()) in
    aerolineasPilotadas->size() <= 2    -- deben ser menos de 2
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (7) `Piloto::maxDosAerolineasPorPiloto': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (7) `Piloto::maxDosAerolineasPorPiloto': FAILED.
```

9. **`Vuelo` → `IdVueloDiferentes`**: asegura que cada vuelo tenga un identificador único. Se evalúa revisando que no haya dos instancias de **`Vuelo`** con el mismo valor de id.

```
inv IdVueloDiferentes:
    Vuelo.allInstances()->isUnique(id)
```

P1_SistemaAviacionTodoBien.soil

```
checking invariant (8) `Vuelo::IdVueloDiferentes': OK.
```

P1_SistemaAviacionTodoMal.soil

```
checking invariant (8) `Vuelo::IdVueloDiferentes': FAILED.
```

10. **`Vuelo` → `PilotosDiferentes`**: establece que un vuelo debe tener un piloto principal y un piloto secundario que sean diferentes. Se comprueba verificando que el piloto secundario no esté definido o que sea distinto del piloto principal.

```
inv PilotosDiferentes:
    self.pilotoSecundario.oclIsUndefined() or self.pilotoPrincipal <> self.pilotoSecundario
```

P1_SistemaAviacionTodoBien.soil

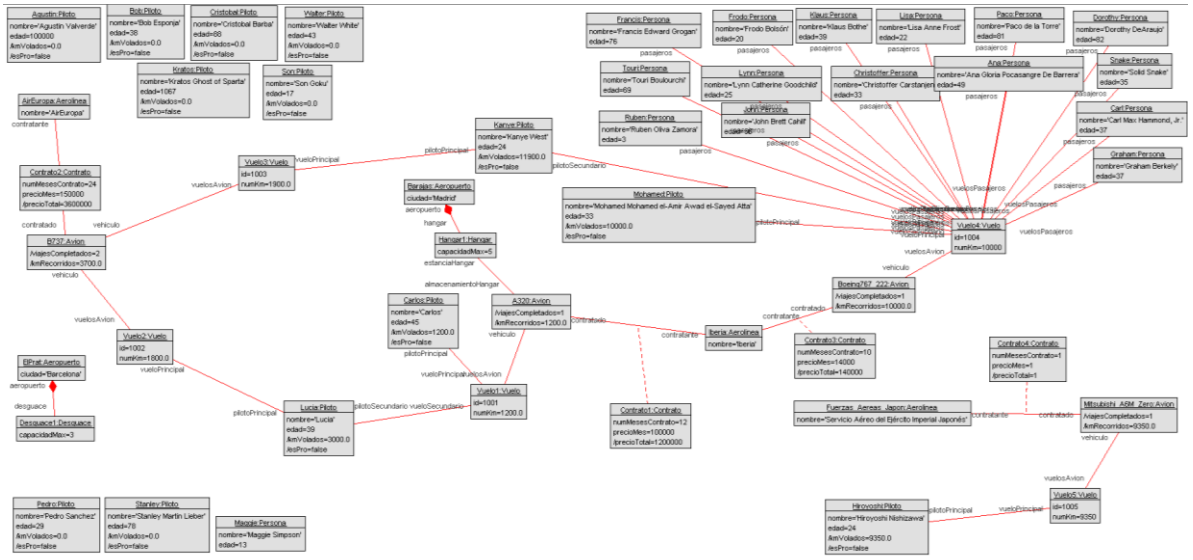
```
checking invariant (9) `Vuelo::PilotosDiferentes': OK.
```

P1_SistemaAviacionTodoMal.soil

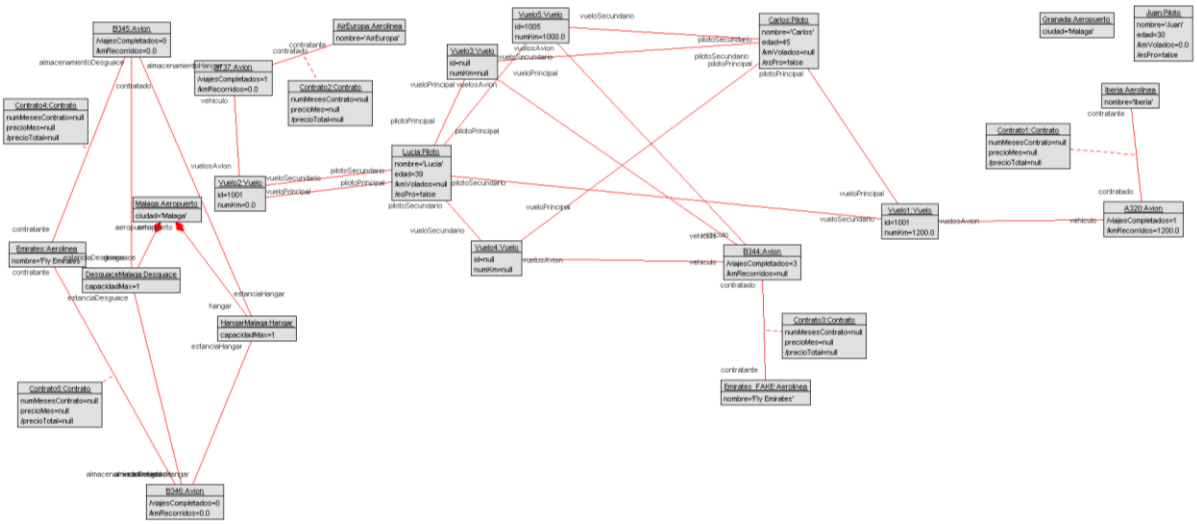
```
checking invariant (9) `Vuelo::PilotosDiferentes': FAILED.
```

2.3.2. Diagramas de objetos (.soil)

Finalmente, los diagramas de objetos de cada uno de los soil utilizado son los siguientes:



3 Diagrama de objetos de P1_SistemaAviacionTodoBien.soil



4 Diagrama de objetos de P1_SistemaAviacionTodoMal.soil

Para probar que no se cumplan los invariantes en el "*5 Diagrama de objetos de P1_SistemaAviacionTodoMal.soil"* no nos importan los valores que tomen ciertos atributos. Por ello, algunos están a null.

Además, se adjuntan también imágenes que confirman que los datos introducidos en ambos soil **no violan ninguna restricción implícita en la multiplicidad de las relaciones** mediante el comando check en consola.

```
P1_SistemaAviacionTodoBien.soil>
use> check
checking structure...
checked structure in 1ms.
checking invariants...
checking invariant (1) `Aerolinea::NombreAerolineaDistinto': OK.
checking invariant (2) `Aeropuerto::CiudadAeropuertoDistinta': OK.
checking invariant (3) `Avion::AvionCorrectamenteDesguace_1000': OK.
checking invariant (4) `Avion::AvionUnicoEstado': OK.
checking invariant (5) `Desguace::CapacidadMaxDesguace': OK.
checking invariant (6) `Hangar::CapacidadMaxHangar': OK.
checking invariant (7) `Piloto::maxDosAerolineasPorPiloto': OK.
checking invariant (8) `Vuelo::IdVueloDiferentes': OK.
checking invariant (9) `Vuelo::PilotosDiferentes': OK.
checked 9 invariants in 0.008s, 0 failures.
use>
```

6 Check en consola de P1_SistemaAviacionTodoBien.soil

```
P1_SistemaAviacionTodoMal.soil>
use> check
checking structure...
checked structure in 1ms.
checking invariants...
checking invariant (1) `Aerolinea::NombreAerolineaDistinto': FAILED.
  -> false : Boolean
checking invariant (2) `Aeropuerto::CiudadAeropuertoDistinta': FAILED.
  -> false : Boolean
checking invariant (3) `Avion::AvionCorrectamenteDesguace_1000': FAILED.
  -> false : Boolean
checking invariant (4) `Avion::AvionUnicoEstado': FAILED.
  -> false : Boolean
checking invariant (5) `Desguace::CapacidadMaxDesguace': FAILED.
  -> false : Boolean
checking invariant (6) `Hangar::CapacidadMaxHangar': FAILED.
  -> false : Boolean
checking invariant (7) `Piloto::maxDosAerolineasPorPiloto': FAILED.
  -> false : Boolean
checking invariant (8) `Vuelo::IdVueloDiferentes': FAILED.
  -> false : Boolean
checking invariant (9) `Vuelo::PilotosDiferentes': FAILED.
  -> false : Boolean
checked 9 invariants in 0.011s, 9 failures.
use>
```

7 Check en consola de P1_SistemaAviacionTodoMal.soil